

Assignment3_Reports_Rhys_Beckett_Final

by Rhys Vaughan Beckett

FILE	25296_RHYS_VAUGHAN_BECKETT_ASSIGNMENT3_REPORTS_RHYS_BECKETT_FINAL_318857_1874422062.PDF (307.56K)		
TIME SUBMITTED	15-JAN-2021 11:38AM (UTC+0000)	WORD COUNT	5471
SUBMISSION ID	142228972	CHARACTER COUNT	32657

Assignment 3: Programming and Software Testing Academic Report Essay

RHYS BECKETT

Table of Contents

Introduction	1
High Cohesion.....	1
Loose Coupling	2
Information Hiding	3
Data Hiding	4
Summary	4
References.....	5

Introduction

This report aims to detail and discuss the value of high cohesion, loose coupling, data hiding and information hiding in software development. It will introduce each principle and explain the importance of upholding their values.

High Cohesion

In his book 'Software Architect's Handbook', Ingenu describes: "Cohesion is the degree to which the elements inside a module belong together. It is the strength of the relationships of elements within a module, and how united they are in their purpose."^[1] In this quote, a module is in reference to the modular programming methodology.

Cohesion is an ordinal measurement that can be considered 'high' or 'low', typically a high cohesion is desired. When a program is considered to be in high cohesion each module should be an independent, interchangeable collection of code that attempt to achieve a singular functionality.

If a module is in low cohesion, containing multiple unrelated routines and functions the overall program becomes harder to understand. Without a logical flow, the developer must follow each step of the program precisely to understand each event that occurs. This causes multiple issues, notably making testing, maintaining and reusing the program that much harder.

There are multiple types of cohesion, these offer varying levels of cohesiveness. From lowest cohesion to highest, these are:

1. Coincidental cohesion,
2. Logical cohesion,
3. Temporal cohesion,
4. Procedural cohesion,
5. Communicational cohesion,
6. Sequential cohesion and

7. Functional cohesion.

Comparing the least cohesive to most, coincidental cohesion occurs when the elements of a module are grouped arbitrarily and there is no relationship between the collected elements. An example of this would be something like a helper class, where multiple functionalities have been grouped for convenience sake. Such cohesion should be avoided, since there is little logic/understanding to grouping these elements together.

Conversely, with the highest cohesion, functional cohesion is desired. It is achieved when all the elements of a module are united in a single, well-defined purpose. Given this environment, the module should be logical and simple to understand. Meaning that testing, maintain and reusing this module should be much easier.

Loose Coupling

When discussing cohesion, coupling is never far. As Martin Fowler said: "There are several ways to describe coupling, but it boils down to this: If changing one module in a program requires changing another module, then coupling exists."^[2] More technically, coupling is the measurement of the relationships between modules. This relationship is measured from loose to tight, or low to high where loose/low coupling is desired.

When the coupling between modules is considered high, it means that those modules are tightly related. In the book Structured Design, Yourdon states: "The more we must know of module B in order to understand module A, the more closely connected A is to B."^[3] This tight relationship implies that A requires B to function, meaning a change to B is likely to break A.

Tightly coupled modules are highly discouraged, due to the illogical flow of data between the modules, they are naturally more complex than their loosely coupled equivalents. This complexity affects multiple areas of development, including increasing the difficulty in maintainability, testing and code reuse.

Similar to cohesion, there are multiple types of coupling that offer varying levels of coupling from tightest to loosest, these are:

1. Content coupling,
2. Common coupling,
3. External coupling,
4. Control coupling,
5. Data-structured coupling,
6. Data coupling,
7. Message coupling and
8. No coupling.

Content coupling is the tightest form of coupling, it occurs when one module directly references the internal/private information 'hidden' in another module. This behaviour is

highly discouraged considering the high likelihood of any change causing a defect in the modules. Then once a defect occurs, the developer might have to scour multiple 'unrelated' modules to find the erroneous code.

As the name suggests, no coupling occurs when two modules have no direct communication and are completely unrelated. This is the technical ideal, but not usually implemented given that some communication between modules is a requirement.

Data coupling is an acceptably loose, more commonly implemented coupling. This type of coupling occurs when two modules share primitive data values through inputs and outputs like method parameters and returns. This type is considered loose since the modules have a very low impact on one another, if one of the modules were to be changed then it is likely the other might not need to be altered to reflect the given change. If a change was required, it should be a minimal change to the method arguments.

Information Hiding

As humorously stated by Steve McConnell: "REWARD for lost software-engineering concept. Responds to the name "information hiding." Last seen in Canada during the late 1970s. Sometimes answers to "encapsulation," "modularity," or "abstraction." [4] In his article, McConnell explains that the definition for information hiding has been confused with those of other similar terms like encapsulation and abstraction, which he considers a shame given the value of the concept.

As a term, information hiding seems to originate from a paper written by David L. Parnas in 1972. In this paper, he wrote: "The second decomposition was made using "information hiding" as a criteria." and "Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings." [5]

Parnas was advocating that system modularization should be based on design decisions. He stated: "We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from others." [5] He further explains that wider system should only access these design decisions through well-defined, mostly unchanging interfaces.

For example, multiple modules within a system might need to handle an ID variable. Without information hiding, those modules might individually access and modify that ID value. In isolation, this scenario might be functional. However, any change to the ID variable could lead to catastrophic failure within the system, where each and every affected module would need to be refactored.

There are a myriad of techniques available for selecting the information to be hidden, many of these techniques are forms of abstraction. Douglas Ross writes: "The essence of abstraction is to extract essential properties while omitting inessential details." [6] Different abstraction techniques will emphasise different information, some techniques include: functional, data, process and object abstraction.

Data Hiding

Once an abstraction technique has been applied to the system information and the specific data to hide has been emphasised, a process must be utilised to actually hide this data. One technique utilised in that process is encapsulation, Wirfs-Brock writes: "The concept of encapsulation as used in an object-orientated context is not essentially different from its dictionary definition. It still refers to building a capsule, in this case a conceptual barrier, around some collection of things." [7]

Many individuals would confuse the definitions of encapsulation and information/data hiding, for example Ian Graham said: "Encapsulation or equivalently information hiding refers to the practice of including within an object everything it needs, and furthermore doing this in such a way that no other object need ever be aware of this internal structure." [8] However, this is incorrect as encapsulation is not inherently required to hide the data contained. As Wirfs-Brock said, encapsulation is simply the process of creating wrappers that collect data together.

This confusion arose due to the fact that these encapsulation wrappers are capable of varying opacities, meaning they can be transparent, allowing any and all other modules within the system accessing the internal data. However, in a majority of scenarios the wrapper is normally defaulted to opaque, therefore most individuals have assumed that encapsulation guarantees data hiding, eventually blending these two definitions subconsciously.

To clarify, data hiding is the process of concealing internal module data, only permitting the functionality within module to access or modify that data. Meaning no external module within the system may directly access this data, instead they are expected to interact through strictly defined interfaces. This process is important as if well implemented, may drastically reduce failure rate and improve code readability, maintainability, testability and reusability.

Summary

Though these four concepts might seem similar given that they each discuss principles relating to good module design, they are uniquely different when closely analysed. Cohesion considers the collective purpose of the functionality contained in each module. Coupling evaluates the relationships between modules, ensuring they do not blend functionalities. Information hiding is a design principle that suggests modules should hide all information that is not required externally. Then lastly, data hiding considers the practical techniques and processes required in concealing the chosen data. Though different, these four concepts all attempt to improve the same development metrics: failure rates, code readability, testability and reusability.

References

1. Ingenu, Joseph. Software Architect's Handbook. Packt Publishing, 2018, p. 168 - 175.
2. Fowler, Martin. Reducing Coupling. IEEE Software. 18 (4), 2001, p. 102 – 104.
<https://martinfowler.com/ieeeSoftware/coupling.pdf> [Accessed: 11/01/2021]
3. Yourdon, Edward; Constantine, Larry L. Structured Design. Yourdon Press, 1978, p. 76.
4. McConnell, Steve. Missing in Action: Information Hiding. IEEE Software. 13 (2), 1996.
<https://stevemcconnell.com/articles/missing-in-action-information-hiding/>
[Accessed: 12/01/2021]
5. Parnas, David L. On the criteria to be used in decomposing systems into modules. Carnegie Mellon University, 1972, p. 17.
6. Ross, Douglas T.; Goodenough, John B.; Irvine, C.A. Software Engineering: Process, Principles, and Goals. IEEE Computer, 8 (5), 1975, p. 22.
7. Wirfs-Brock, Rebecca; Wilkerson, Brian; Wiener, Lauren. Designing object-oriented software. Englewood Cliffs, New Jersey: Prentice-Hall, 1990, p. 18.
8. Graham, Ian. Object-Oriented Methods. Reading, Massachusetts: Addison-Wesley, 1991.

Assignment 3: Database Systems Academic Report

RHYS BECKETT

Table of Contents

Introduction	7
Transaction Processing	7
ACID	8
Atomicity	8
Consistency.....	8
Isolation.....	9
Durability.....	9
Summary	10
References.....	10

Introduction

This report aims to detail the importance of transaction processing within database systems, it will discuss relevant topics including: the importance of maintaining the ACID transaction properties, and the solutions utilised by database management systems to ensure data correctness in a concurrently accessible database.

Transaction Processing

Transaction processing is a form of information processing that describes the tools and techniques utilised in handling a myriad of individual transactions. In the book Principles of Transaction Processing, Phillip Bernstein compares a database transaction to a business transaction. He states: "A business transaction is an interaction in the real world, usually between an enterprise and a person or another enterprise, where something is exchanged. For example, it could involve exchanging money, products, information, or service requests." [1] As explained, a business transaction can be the exchange of product between two parties. This exchange can be described in multiple logical steps, for example:

1. Customer requests to purchase a product,
2. Business confirms available stock,
3. Business receives the money,
4. Customer receives the product.

Though these steps may seem simple, in a large database schema they could be considered failure/error-prone. In the given example, two customers might be attempting to purchase the same product within the same timeframe. Normally, this would not be an issue. However, in the scenario that the business only has a single instance of the desired stock, following the previous steps both customers would have paid for product yet one customer would be left without the promised product. Transaction Processing attempts to avoid such

scenarios by applying strict rules, tools and techniques to managing these concurrent transactions.

ACID

ACID describes a set of properties that a database transaction should maintain in an effort to guarantee data validity. As coined by Haerder & Reuter in their paper "Principles of transaction-oriented database recovery"[2] these are: Atomicity, Consistency, Isolation and Durability.

Atomicity

When in reference to a transaction, atomicity describes the behaviour of the transaction process regarding the success of the given transaction. To further quote Bernstein: "First, a transaction needs to be **atomic** (or **all-or-nothing**), meaning that it executes completely or not at all. There must not be any possibility that only part of a transaction program is executed." [1]

By adopting this polarised attitude, the database management systems can confidently attempt any transaction understanding that the relevant operations cannot produce wrongful data through an incomplete process.

For example, the transfer of currency between two bank accounts: this transaction includes two simple steps, the withdrawal and deposit of money. Due to a myriad of issues the database management system may withdraw the money yet fail in depositing the money. Without atomicity, that currency is lost, and the data stored in the database is no longer accurate. However, if atomicity is applied the DBMS will recognise the failure within the operation and cancel the entire transaction, utilising its recovery systems to revert the previous changes to their original state.

Though atomicity is vital for any reliable DBMS, it does not solve any concurrency issues. For example, if a bank account contains £100.00 and receives two simultaneous transactions to withdraw £100.00 both atomic operations will succeed. From the perspective of both the transaction operations, there is £100.00 in the bank account, so they'll both successfully withdraw the 'same' £100.00. This issue is solved by the isolation property, which will be discussed further in the report.

Consistency

As described in 'Transaction Processing: Concepts and Techniques', the consistency property is fulfilled when "A transaction produces consistent results only; otherwise it aborts. A result is consistent if the new state of the database fulfils all the consistency constraints of the application." [3] Meaning a transaction is only considered successful if the updated database state fulfils the given constraints, these include constraints like: primary keys, foreign keys, and check constraints.

A transaction consists of multiple operations, checking that the transaction is maintaining consistency throughout these operations would be inefficient and expensive. Instead, these database management systems utilise the final transaction commit as a guarantee that the new database state is consistent. If the new database state is not consistent, meaning it failed to satisfy a constraint; then the final commit operation will fail. Given that the DBMS utilises the ACID properties, then the full transaction will fail due to the atomicity property and the system will recover/rollback to the initial database state.

Whilst atomicity, isolation and durability are properties that are ensured by the database management system, consistency is jointly ensured by both the DBMS and the individual transaction programs. Meaning it is the responsibility of the application's programmer to ensure that the transaction programs they develop result in a consistent database state.

Consistency is an important property for any database management system, however ensuring consistency does not improve concurrent functionality. Concurrent transactions can be consistent yet still cause other unforeseen issues.

Isolation

In his book 'Transactional Information Systems', Weikum describes: "A transaction is isolated from other transactions, in the sense that each transaction behaves as if it were operating alone with all resources to itself."^[4] From a technical viewpoint, this isolation behaviour is comparable to serialization. Meaning, that the isolation property is upheld when the transactions are seemingly executed in series, one after another with no overlap.

Actually, running all transaction in series would be incredibly inefficient and expensive, instead the database management system provides an illusion of serialization by locking currently accessed data. This lock ensures that a later transaction cannot access data that has the potential to mutate, forcing it to wait until the previous transaction is completed and the lock is removed. By utilising the behaviour, the DBMS can run multiple concurrent transactions whilst appearing serial to an individual user, since each transaction should be accessing independent data.

The issue previously described in the banking scenario would be solved by upholding this property. Though two transactions would attempt to withdraw £100.00 from the bank account, the first transaction would lock the data relevant to the account. Since this data is locked, the second transaction would be forced to wait. Once the first transaction is successfully completed, the lock is released. Since the previous transaction cleared the remaining currency from the account, the second transaction will fail due to the withdrawal operation causing an inconsistent state of the database.

Durability

Haerder writers: "Durability. Once a transaction has been completed and committed its results to the database, the system must guarantee that these results survive any subsequent malfunctions."^[2] Unless explicitly overwritten or deleted, the results of a successful transaction should be immutable and stable, regardless of the time since last

transaction. Meaning the database state should survive any subsequent failure, be it user, environment, software or hardware in nature.

Some transactions reflect a contractual exchange between individuals, like the exchange of currency or services. These transactions are only possible due to an inherent trust in the system. Durability is a vital property required in maintaining this trust. Once the transaction between individuals is confirmed, they trust that this result is permanent.

Database management systems attempt to uphold durability by maintaining a copy of a transaction's every update to a log file whilst the transaction program is executing. Once the commit operation has been called, the system will first confirm that these updates are applied to the database and written to stable storage, then notify the user of a successful commit.

If ever the database system were to fail for any reason, be it power-failure or system malfunction; the log record can be utilised to repair any 'damage' to the database. By comparing the database state against the log of confirmed transactions, the system can locate corrupted data, then apply the transactions to recover the original database state.

Though durability does not inherently solve any concurrency issues, a concurrent database system is more susceptible to failure. In this regard, good transaction durability is a requirement for any concurrent database management system that handles sensitive data.

Summary

In summary, the transaction process is a vital area of database systems. Though coined in 1983, the ACID properties are still relevant to this day. By upholding these properties, the database management system can guarantee a reliable, trustworthy service. Though more complex, even concurrent systems should strive to match these standards given the importance of maintaining a stable service, especially when processing simultaneous transactions.

References

1. Bernstein, Phillip A.; Newcomer, Eric. Principles of Transaction Processing. Elsevier Science, 2009, p. 10 - 15.
2. Haerder, Theo; Reuter, Andreas. Principles of transaction-oriented database recovery. ACM Computing Surveys, 1983, 15(14), p. 289 - 290.
3. Gray, Jim; Reuter, Andreas. Transaction Processing: Concepts and Techniques. Elsevier Science, 1992, p. 166 - 167.
4. Weikum, Gerhard; Vossen, Gottfried. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Elsevier Science, 2002, p. 24 - 25.

Assignment 3: Programming and Software Testing Reflective Report

RHYS BECKETT

Introduction

This reflective report aims to discuss the design decisions I made during this assignment whilst considering any interesting features implemented. It will examine the system modularization, function overloading, input validation, prepared statements and unit testing.

System Modularization

Honestly, without a team to work around, the design of this system was more organic than it probably should have been. Despite this, I do believe that the modules make logical sense, especially once I refactored them after writing the academic report. Notably, the system is divided into three primary concepts, being: SQL functionality, user input, and validation.

Firstly, the SQL functionality is separated into multiple files, like 'sqlInsertFunctions', 'sqlDeleteFunctions' and 'sqlUpdateFunctions'. Obviously, they are separated by the functionality of the different SQL queries available. This would seem to imply functional cohesion, however given that these files are filled with individual different tasks that utilise the same SQL queries, it is more likely to be logical cohesion.

Originally, most of the user input functionality was developed within the 'userController' file. I quickly realised that there was too much functionality to be retained in a single module, therefore I separated the validation code into a separate file titled 'validationFunctions'. Similar to the SQL functionality, this module separation only reaches logical cohesion given that all these validation functions do not attempt to resolve a singular task.

Even after separating the validation functionality from the 'userController' I further divided the file into smaller modules, these include the files: 'chooseAddress', 'chooseDriver' and 'chooseVehicle'. Though these modules are still mostly logical cohesion, this separation does narrow the range of functionality that each module handles, bringing the cohesion closer to the desired functional cohesion.

Most of these modules ascribe to logical cohesion, which is a relatively low form of cohesion. To achieve high cohesion, it would be better to utilise functional cohesion where each module is solely focused on resolving a singular task. However, given the nature of the SQL functionality this would probably result in a lot of duplicated query code.

Despite the cohesion issues, the relationships between these modules should mostly be data coupling since they interact through strict, function interfaces that only pass primitive data between one-another. Therefore, I believe that the design is an acceptable solution in this situation. Hopefully, future learning will offer more elegant object-orientated solutions.

Function Overloading

Within the SQL themed modules, I heavily utilised a technique called 'function overloading'. This technique allows multiple functions to share the same name whilst utilising different constructors. There are a large number of potential permutations to the data passed through an SQL query. Without function overloading, I would have to give each individual permutation a unique name. Instead, this technique allows the function 'selectStatement()' to be passed a wide array of different data types as the filtering in the SQL 'WHERE' clause. This probably reduced the potential for human-error in the code given that I had a lot less function names to remember.

Input Validation

Much of my time was spent developing the input validation functionality, ensuring that the program runs smoothly despite erroneous user input. There are functions to test every data type utilised in the database, including: intervals, strings (registration plates, phone numbers, postcode, etc.), integers, dates and entities. Since we are validating a user input each data type begins as a string, to validate the different data types I utilised the try/catch format. For example, I catch 'NumberFormatException' when I fail to parse the string value into an integer, then return a false Boolean. These validators are called within while loops, that will constantly iterate until the user enters a valid input or exits the current task.

Prepared Statements

The SQL functionality utilises prepared statements to format the queries. This is an alternative to string concatenation, which is susceptible to a hacking technique called SQL-Injection. This hacking technique replaces the string values with further SQL queries, leading to queries the developer did not intend. Fortunately, prepared statements minimise these issues by ensuring the data inserted into the query is relevant and correct. Since the prepared statement insertion code varies depending on the given data types, this technique is complimented by the function overloading process.

Unit Testing

I believe that I have fabricated an extensive variety of unit tests, 45 in all for this project, the majority of which test the SQL and validation functionality. This is mostly because those functionalities produce definitive returns, which are a requirement for the comparative assertions.

Unfortunately, a large area of functionality is not unit tested because it either utilises user input or focuses on console printing, instead I needed to thoroughly manual test. If given another opportunity, I would hope to learn and implement unit tests for both these functionalities.

Summary

In summary, though I still have much more left to learn, I believe that I've fully utilised my current knowledge to produce a system that satisfies a majority of the given specification requirements. I've applied much of the learning from the module to offer a relatively elegant solution, notably being the extensive whitebox unit testing, database handling scripts and the logic utilised in user input/output.

Assignment 3: Database Systems Reflective Report

RHYS BECKETT

Introduction

In this report I will reflect on the database design of the system utilised in this assignment. I will consider the architecture of each table, the relationships between the tables, the various applied constraints and lastly discuss both the fabricated indexes and the views that were utilised, evaluating their impact in improving query performance.

Tables

Whilst designing the initial architecture of the database, I iterated through a couple designs.

Though the first design did meet the requirements of the specification, I realised it was out of scope and overly complicated:

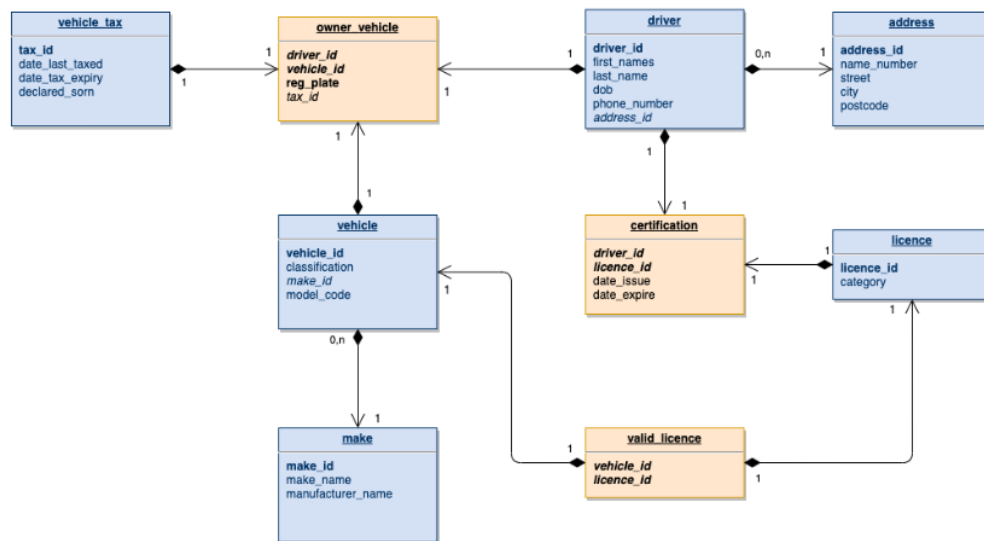


Figure 1 - Initial Database Design Draft

As illustrated above, this design contains multiple data concepts that are not mentioned within the specification; most notable being the driver licensing. Given that this added complexity would be considered a failing in a commercial environment, I chose to attempt a second design draft.

However, despite this failing, I do believe that this design utilised good principles and would be an effective solution to a more ambitious project. Notably, the utilisation of multiple bridging tables to achieve the 3rd Normal Form.

When designing the second draft of the database, I was stricter about achieving the specification requirement with minimal design, given that a larger database would simply be more inefficient and expensive to maintain. With these principles in mind, I created this design:

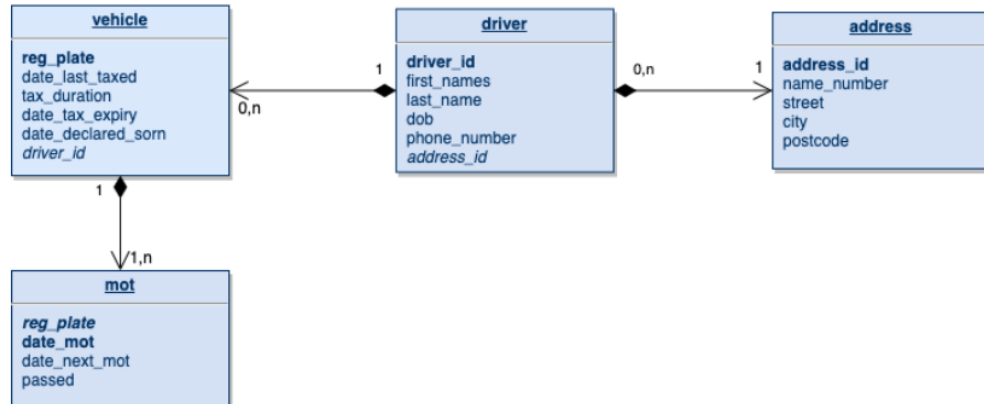


Figure 2 - Final Database Design

I believe that this design achieves a good balance between satisfying the specification requirement and maintaining a minimal database. Compared to the original draft, this database contains less than half the tables, which should improve the theoretical performance of the system.

Notable reductions to the system were the removal of the driver licensing concept, and the modification of the vehicle entity. Given that this database was to be utilised in a road tax system, I realised that information regarding the vehicle models was erroneous. Instead, the vehicle entity should be a store of tax information.

The specification lists a couple relational requirements, these are:

- It must be possible for a driver to own multiple cars and for multiple drivers to live at the same address, with an appropriate database design to accommodate.

I understood these requirements to mean, there must be a 'One-to-Many' relationship from driver to vehicle and similarly from address to driver. Since these were not 'Many-to-Many' relationships, bridging tables were not required. Instead, I simply utilised some referential integrity constraints (foreign keys).

Within a 'One-to-Many' relationship a foreign key is added as a column within the 'Many' entities and references an existing column with the 'One' entity. In this design the utilised foreign keys include: 'driver_id' in vehicle, 'address_id' in driver and 'reg_plate' in mot.

Constraints

There are multiple types of constraints available in database design, to satisfy the requirements this specific design utilises primary key, foreign key and check constraints.

Both the driver_id and address_id are integer based primary keys that if left empty, will automatically populate the column with the next highest value. This was done by specifying their datatype to be 'serial'.

In contrast, reg_plate is a string based primary key. I chose to assign the registration plate as the primary key given the context of the entity, being that no two vehicles may share the same registration plate. Since this value was already unique, it would have been inefficient to create another unique column as the primary key for the table.

Lastly, the mot table utilises a composite primary key that utilises both 'reg_plate' and 'date_mot'. These work as a unique identifier given the assumption that no vehicle would have two MOT appointments within the same day, which is logical given that there is normally a 12-month gap between these appointments.

The mot table was actually fabricated simply to satisfy the requirement for a composite primary key. This was not ideal, given that the mot entity is not a technical requirement of the system. I considered creating a composite primary key on the (name_number, postcode) columns within address instead. However, I decided against this action since the driver table needed to reference the address table via a foreign key. Since tables that reference a composite foreign key must include both columns, I decided this was inefficient duplication of data.

Though the specification only asks for 3 check constraints to be implemented, this database utilises more than double that amount. These include:

- Driver table: a check on the dob column to confirm the driver is over 16 years of age.
- Driver table: a check on the phone_number to confirm it is a string of 11 integer characters.
- Vehicle table: a check on date_tax_expiry to confirm it is later than the date_last_taxed.
- Vehicle table: a check on tax_duration to confirm the interval is either 6 or 12 months long.
- Vehicle table: a check on date_last_taxed and date_declared_sorn to confirm they are not in the future.
- MOT table: a check on mot_date to confirm it also is not in the future.

Though more constraints could have been applied, I believe these to be a suitable amount. Some constraints can cause more harm than good, especially on data with diverse formats like names and addresses.

Lastly, the database includes two generated constraints, specifically on the date_tax_expiry column and date_next_mot columns in vehicle and mot tables respectively. Specifically, these constraints calculate a date comparing the date of one column against the interval within another. By utilising this constraint, the database management system can automatically calculate the column value on insert and update. This is beneficial since it removes a potential for human error, making the system more robust and reliable.

Indexes

Indexing is a powerful database technique that can significantly improve the query performance of the system. When a query searches against a table value without an index, it is iterating over an unordered list. Since there is no logic to the sequence of data, the system must brute force compare against every individual value in the table. However, by applying an index, the database system now has an ordered reference to the data. Given that this reference is an ordered list, a query can utilise a variety of efficient search algorithms to return the desired results in an incredibly timely manner.

Though indexing can significantly improve query performance, it is not perfect. As previously mentioned, an index creates a reference to the data within a given table. This reference must be updated each time the table is modified, meaning each index adds small over-head to the database performance. Therefore, a database design should attempt to balance the benefits of indexing against the performance cost.

With this in mind, I created secondary indexes only against the data most often queried in the software. These include indexes for:

- first_names and last_name DESC in the driver table,
- postcode DESC in the address table,
- date_declared_sorn DESC in the vehicle table and
- date_tax_expiry DESC in the vehicle table

Though there are some other queries utilised by the program, I decide they were too niche and infrequently utilised to justify an index.

Views

From an external perspective, a view is simply another table. However, in reality they are often the aggregate of multiple actual tables. Similar to indexes, they are a technique to be utilised in moderation. A couple can drastically improve query performance, too many can reduce the overall performance of the database system.

I chose to create a view for the specification requirement:

- It must be possible to view drivers whose road tax is close to expiry through the program.

Unlike other requirements, the parameters of the query that satisfy this requirement are static. Therefore, it is a suitable candidate for fabricating a view as the logic utilised to produce the resulting table would rarely require modification. Once fabricated, the query for that data is significantly faster given that it is no longer required to perform any searching or calculations, instead it is simply required to retrieve the data.

Summary

Overall, I have learnt a lot throughout this piece of work and believe that I've satisfied all the specification requirements.

Though the process required multiple iterations, the design produced is an elegant solution to the brief that also achieves 3rd Normal Form. If given another opportunity, I would consider removing the 'mot' table given that it was not utilised in the software. However, I would need a solution to the composite primary key requirement.

The implementation of the database was very successful, by utilising multiple, powerful tools and techniques I created an efficient solution to the given requirements. This was evident as I rarely faced any issues from the database whilst developing the software code.

Assignment3_Reports_Rhys_Beckett_Final

GRADEMARK REPORT

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

Programming and Software Testing mark: 83

Database Systems mark: 89

Implementation (50): 44

Programming and Software Testing Report (50): 39

Database Systems Report (50): 45

A very high quality submission overall. The implementation is robust - clearly a good bit of time and effort went into it.

The reports are well written. I know you're aware you have a little bit of work to do on your academic practice, but overall these were quite a bit better than the previous submission.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21
